Ph. D. Thesis

Design, Evaluation and Analysis of Combinatorial Optimization Heuristic Algorithms

Author Daniil Karapetyan
Supervisor Prof. Gregory Gutin

Department of Computer Science
Royal Holloway College
University of London

July 2010

Declaration

The work presented in this thesis is the result of original research carried out by myself. This work has not been submitted for any other degree or award in any other university or educational establishment.

Daniil Karapetyan

Abstract

Combinatorial optimization is widely applied in a number of areas nowadays. Unfortunately, many combinatorial optimization problems are NP-hard which usually means that they are unsolvable in practice. However, it is often unnecessary to have an exact solution. In this case one may use heuristic approach to obtain a near-optimal solution in some reasonable time.

We focus on two combinatorial optimization problems, namely the Generalized Traveling Salesman Problem and the Multidimensional Assignment Problem. The first problem is an important generalization of the Traveling Salesman Problem; the second one is a generalization of the Assignment Problem for an arbitrary number of dimensions. Both problems are NP-hard and have hosts of applications.

In this work, we discuss different aspects of heuristics design and evaluation. A broad spectrum of related subjects, covered in this research, includes test bed generation and analysis, implementation and performance issues, local search neighborhoods and efficient exploration algorithms, metaheuristics design and population sizing in memetic algorithm.

The most important results are obtained in the areas of local search and memetic algorithms for the considered problems. In both cases we have significantly advanced the existing knowledge on the local search neighborhoods and algorithms by systematizing and improving the previous results. We have proposed a number of efficient heuristics which dominate the existing algorithms in a wide range of time/quality requirements.

Several new approaches, introduced in our memetic algorithms, make them the state-of-the-art metaheuristics for the corresponding problems. Population sizing is one of the most promising among these approaches; it is expected to be applicable to virtually any memetic algorithm.

In memory of my mother

Acknowledgment

I would like to thank my supervisor Gregory Gutin for his patience, encouragement and advice in all aspects of academic life over the last three years.

I would also like to thank my family and, especially, my grandfather who was continuously inspiring my interest in technical sciences for at least two decades.

This research was supported by the Computer Science Department of the Royal Holloway College and a Thomas Holloway Scholarship.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Nowadays combinatorial optimization problems arise in many circumstances, and we need to be able to solve these problems efficiently. Unfortunately, many of these problems are proven to be NP-hard, i.e., it is often impossible to solve the instances in any reasonable time.

However, in practice one usually does not need an exact solution of the problem. In this case one can use a heuristic algorithm which yields a near-optimal solution in a satisfactory time. Some of the heuristics, so-called approximation algorithms, guarantee certain solution quality and the polynomial running time. Unfortunately, this nice theoretical property is usually achieved at the cost of relatively poor performance. In other words, a simple heuristic is often faster and yields better solutions than an approximation algorithm, though a simple heuristic does not guarantee any quality and in certain cases it yields very bad solutions.

In this research we focus on heuristic algorithms which usually have no guaranteed solution quality. We are interested in design and selection of the most efficient algorithms for real-world use and, thus, we pay a lot of attention to experimental evaluation.

As a case study we consider two combinatorial optimization problems: the Generalized Traveling Salesman Problem (GTSP) and the Multidimensional Assignment Problem (MAP). Both problems are known to be NP-hard and each has a host of applications.

Though both GTSP and MAP are very important, the researchers did not pay enough attention to certain areas around these problems. In particular,

the literature lacks any thorough surveys of GTSP or MAP local search; there are just a few metaheuristics for MAP and all of them are designed for only one special case of the problem; there exists no standard test bed for MAP which would include a wide range of instances of different sizes and types. This work fills many of these and some other gaps, and, moreover, some of the ideas proposed here can be applied to many other optimization problems.

We start from a thorough explanation of the case study problems, their applications and existing solution methods.

Chapter 2 is devoted to some approaches in heuristic design. It discusses test bed construction, shows several examples on how theoretical tools can help in design of practically efficient heuristics and provides a set of advices on high-performance implementation of an algorithm.

Chapter 3 introduces a classification of GTSP neighborhoods, proposes several new ones and includes a number of algorithms and improvements which significantly speed up exploration of these neighborhoods both theoretically and practically. Special attention is paid to adaptation for GTSP of the well-known Lin-Kernighan heuristic, originally designed for the Traveling Salesman Problem.

Chapter 4, similar to Chapter 3, considers the MAP neighborhoods and local search algorithms. It splits all the MAP neighborhoods into two classes, generalizes the existing approaches, proposes some new ones and, finally, considers a combined local search which explores neighborhoods of both types together. An extensive experimental analysis is intended to select the most successful heuristics.

Chapter 5 is devoted to the so-called Memetic Algorithms (MA). MA is a kind of evolutionary algorithms which applies an improvement procedure to every candidate solution. Several evolutionary algorithms for GTSP, including MAs, are already presented in the literature. We propose a new MA which features a powerful local search and an efficient termination criterion. It also uses some other improvements like variation of the population size according to the instance size. In our experiments, this algorithm clearly outperforms all GTSP metaheuristics known from the literature with respect to both solution quality and running time.

We develop the idea of adaptive population size and apply it to MAP. In our new MA, we use a time based termination criterion, i.e., the algorithm is given some certain time to proceed. The population size is selected to exploit the given time with maximum efficiency. The experimental results provided in Chapter 5 show that the designed algorithm is extremely flexible in solution quality/running time trade-off. In other words, it works efficiently for a wide range of given times.

Most of the experimental results are presented in detailed tables and placed in Appendix A.

We have already published many results provided here. Some aspects of heuristic design are discussed in [37, 36, 40, 62]. The test bed for MAP was developed in [60, 62]. The Lin-Kernighan heuristic is adapted for GTSP in [58]. Other GTSP local searches are presented in [59]. A similar discussion of MAP local search can be found in [38] and [60]. The memetic algorithm for GTSP is proposed in [37]. The population sizing and the memetic algorithm for MAP are suggested in [39] and [61].

1.1 Generalized Traveling Salesman Problem

The Generalized Traveling Salesman Problem (GTSP) is an extension of the Traveling Salesman Problem (TSP). In GTSP, we are given a complete graph $G = (V, E)$, where V is a set of n vertices, and every edge $x \to y \in E$ is assigned a weight $w(x \to y)$. We are also given a proper partition of V into clusters $C_1, C_2, \ldots, C_m$, i.e., $C_i \cap C_j = \varnothing$ and $\bigcup_i C_i = V$. A feasible solution, or a *tour*, is a cycle visiting exactly one vertex in every cluster. The objective is to find the shortest tour.

There also exists a variation of the problem where the tour is allowed to visit a cluster more than once, see, e.g., [25]. However, this variation is equivalent if the weights in the graph G satisfy the triangle inequality $w(x \to y) \leq w(x \to z \to y)$ for any $x, y, z \in V$. In what follows, we consider the problem of finding the shortest cycle which visits *exactly* one vertex in each cluster.

If the weight matrix is symmetric, i.e., $w(x \to y) = w(y \to x)$ for any

$x \in V$ and $y \in V$, the problem is called *symmetric*. Otherwise it is an *asymmetric* GTSP.

There are many publications on GTSP (see, e.g., the surveys [23, 34]) and the problem has many applications in warehouse order picking with multiple stock locations, sequencing computer files, postal routing, airport selection and routing for courier planes and some others, see, e.g., [24, 25, 71, 78] and references there.

The problem is NP-hard, since the *Traveling Salesman Problem* (TSP) is a special case of GTSP when $|C_i| = 1$ for each i. GTSP is trickier than TSP in the following sense: it is an NP-hard problem to find a minimum weight collection of vertex-disjoint cycles such that each cluster has exactly one vertex in the collection (and the claim holds even when each cluster has just two vertices) [43]. Compare it with the well-known fact that a minimum weight collection of vertex-disjoint cycles covering the whole vertex set in a weighted complete digraph can be found in polynomial time [42].

1.1.1 Additional Notation

In what follows we use the following notation:

- s is the maximum cluster size. Obviously $\lceil n/m \rceil \le s \le n - m + 1$.
- γ is the minimum cluster size. Obviously $1 \le \gamma \le \lfloor n/m \rfloor$.
- $Cluster(x)$ is the cluster containing the vertex x.
- $w(x_1 \to x_2 \to \ldots \to x_k)$ is the weight of a path $x_1 \to x_2 \to \ldots \to x_k$, i.e., $w(x_1 \to x_2 \to \ldots \to x_k) = w(x_1 \to x_2) + w(x_2 \to x_3) + \ldots + w(x_{k-1} \to x_k)$.
- $w_{\min}(X \to Y) = \min\limits_{x \in X, y \in Y} w(x \to y)$ denotes the minimum weight of an edge from a vertex set X to a vertex set Y. If one substitutes a vertex v instead of, e.g., X then we assume that $X = \{v\}$. Function $w_{\max}(X \to Y)$ is defined similarly.
- T_i denotes the vertex at the ith position in the tour T. We assume that $T_{i+m} = T_i$.

- Tour T is also considered as a set of its edges, i.e., $T = \{T_1 \to T_2,\ T_2 \to T_3,\ \ldots,\ T_{m-1} \to T_m,\ T_m \to T_1\}$.

- $\mathit{Turn}(T, x, y)$ denotes a tour obtained from T by replacing the fragment $T_{x+1} \to T_{x+2} \to \ldots \to T_y$ with $T_y \to T_{y-1} \to \ldots \to T_{x+1}$:

$$\begin{aligned}\mathit{Turn}(T, x, y) = T_x \to T_y \to T_{y-1} \to T_{y-2} \to \ldots \to T_{x+1} \\ \to T_{y+1} \to T_{y+2} \to \ldots \to T_{x-1} \to T_x\,.\end{aligned}$$

  Observe that for the symmetric GTSP the $\mathit{Turn}(T, x, y)$ tour can be obtained by deleting the edges $T_x \to T_{x+1}$ and $T_y \to T_{y+1}$ and adding the edges $T_x \to T_y$ and $T_{x+1} \to T_{y+1}$:

$$\mathit{Turn}(T, x, y) = T \setminus \{T_x \to T_{x+1},\ T_y \to T_{y+1}\} \cup \{T_x \to T_y,\ T_{x+1} \to T_{y+1}\}$$

  and, hence, the weight of the obtained tour is as follows:

$$\begin{aligned}w(\mathit{Turn}(T, x, y)) = w(T) - w(T_x \to T_{x+1}) - w(T_y \to T_{y+1}) \\ + w(T_x \to T_y) + w(T_{x+1} \to T_{y+1})\,. \quad (1.1)\end{aligned}$$

1.1.2 Existing Approaches

Various approaches to GTSP have been studied. There are exact algorithms such as branch-and-bound and branch-and-cut algorithms in [25]. While exact algorithms are very important, they are unreliable with respect to their running time that can easily reach many hours or even days. For example, the well-known TSP solver Concorde [5] can easily solve some TSP instances with several thousand cities, but it could not solve several asymmetric instances with 316 cities within the time limit of 10^4 s [25].

Several researchers [11, 73, 79] proposed transformations of GTSP into TSP. At the first glance, the idea to transform a little-studied problem into a well-known one seems to be natural. However, this approach has a very limited application. Indeed, it requires exact solutions of the obtained TSP instances because even a near-optimal solution of such TSP may corre-

spond to an infeasible GTSP solution. At the same time, the produced TSP instances have quite unusual structure which is hard for the existing TSP solvers. A more efficient way to solve GTSP exactly is a branch-and-bound algorithm designed by Fischetti et al. [25]. This algorithm was able to solve instances with up to 89 clusters. Two approximation algorithms were proposed in the literature, however, both of them are unsuitable for the general case of the problem, and the guarantied solution quality is unreasonably low for the real-world applications, see [12] and references therein.

In order to obtain good (but not necessary exact) solutions for larger GTSP instances, one should use heuristic approach. Several construction heuristics and local searches were discussed in [12, 41, 51, 92, 96] and some others. A number of metaheuristics were proposed in [12, 53, 87, 95, 96, 101, 103].

1.2 Multidimensional Assignment Problem

The *Multidimensional Assignment Problem* (MAP), abbreviated s-AP in the case of s dimensions and also called *(axial) Multi Index Assignment Problem* (MIAP) [8, 83], is a well-known optimization problem. It is an extension of the *Assignment Problem* (AP), which is exactly the two dimensional case of MAP. While AP can be solved in polynomial time [69], s-AP for every $s \geq 3$ is NP-hard [28] and inapproximable [15][1], i.e., there exists no k-approximation algorithm for any fixed k.

The most studied case of MAP is the case of three dimensions [1, 4, 7, 19, 52, 98] though the problem has a host of applications for higher numbers of dimensions, e.g., in matching information from several sensors (data association problem), which arises in plane tracking [77, 84], computer vision [102] and some other applications [4, 8, 13], in routing in meshes [8], tracking elementary particles [88], solving systems of polynomial equations [10], image recognition [32], resource allocation [32], etc.

For a fixed $s \geq 2$, the problem s-AP is stated as follows. Let $X_1 = X_2 = \ldots = X_s = \{1, 2, \ldots, n\}$; we will consider only vectors that belong to the

[1]Burkard et al. show it for a special case of 3-AP and since 3-AP is a special case of s-AP the result can be extended to the general MAP.

Cartesian product $X = X_1 \times X_2 \times \ldots \times X_s$. Each vector $e \in X$ is assigned a non-negative weight $w(e)$. For a vector $e \in X$, the component e_j denotes its jth coordinate, i.e., $e_j \in X_j$. A collection A of $t \leq n$ vectors $A^1, A^2, \ldots, A^t$ is a *(feasible) partial assignment* if $A^i_j \neq A^k_j$ holds for each $i \neq k$ and $j \in \{1, 2, \ldots, s\}$. The *weight* of a partial assignment A is $w(A) = \sum_{i=1}^{t} w(A^i)$. A partial assignment with n vectors is called *assignment*. The objective of s-AP is to find an assignment of minimal weight.

A graph formulation of the problem (see Fig. 1.1) is as follows. Having a complete s-partite graph G with parts X_1, X_2, ..., X_s, where $|X_i| = n$, find a set of n disjoint cliques in G, each of size s, of the minimal total weight with every clique Q in G assigned a weight $w(Q)$ (note that in the general case $w(Q)$ is not simply a function of the edges of Q).

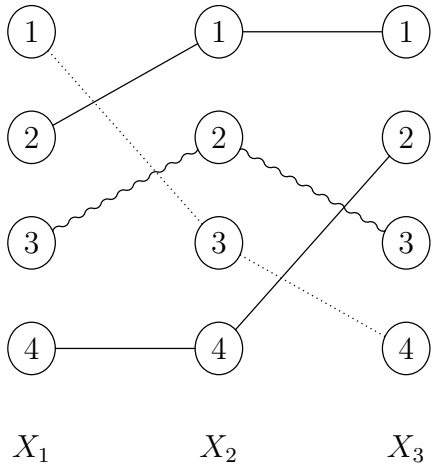

Figure 1.1: An example of an assignment for a MAP with $s = 3$ and $n = 4$. This assignment contains the following vectors: (1, 3, 4), (2, 1, 1), (3, 2, 3) and (4, 4, 2). Note that to simplify the picture we show only a subset of edges for every clique.

We also provide a *permutation form* of the assignment which is sometimes very convenient. Let $\pi_1, \pi_2, \ldots, \pi_s$ be permutations of $X_1, X_2, \ldots, X_s$, respectively. Then $\pi_1\pi_2 \ldots \pi_s$ is an assignment of weight $\sum_{i=1}^{n} w(\pi_1(i)\pi_2(i) \ldots \pi_s(i))$. It is obvious that some permutation, say the first one, may be fixed without any loss of generality: $\pi_1 = 1_n$, where 1_n is the identity permutation of n elements. Then the objective of the problem is as follows:

$$\min_{\pi_2,\ldots,\pi_s} \sum_{i=1}^{n} w(i\pi_2(i) \ldots \pi_s(i))$$

and it becomes clear that there exist $n!^{s-1}$ feasible assignments and the fastest known algorithm to find an optimal assignment takes $O(n!^{s-2}n^3)$ operations. Indeed, without loss of generality set $\pi_1 = 1_n$ and for every combination of $\pi_2, \pi_3, \ldots, \pi_{s-1}$ find the optimal π_s by solving corresponding AP in $O(n^3)$.

Thereby, MAP is very hard; it has n^s values in the weight matrix, there are $n!^{s-1}$ feasible assignments and the best known algorithm takes $O(n!^{s-2}n^3)$ operations. Compare it, e.g., with the Travelling Salesman Problem which has only n^2 weights, $(n-1)!$ possible tours and which can be solved in $O(n^2 \cdot 2^n)$ time [48].

Finally, an integer programming formulation of the problem is as follows.

$$\min \sum_{i_1 \in X_1, \ldots, i_s \in X_s} w(i_1 \ldots i_s) \cdot x_{i_1 \ldots i_s}$$

subject to

$$\sum_{i_2 \in X_2, \ldots, i_s \in X_s} x_{i_1 \ldots i_s} = 1 \qquad \forall i_1 \in X_1 \,,$$

$$\ldots$$

$$\sum_{i_1 \in X_1, \ldots, i_{s-1} \in X_{s-1}} x_{i_1 \ldots i_s} = 1 \qquad \forall i_s \in X_s \,,$$

where $x_{i_1 \ldots i_s} \in \{0, 1\}$ for all $i_1, \ldots, i_s$ and $|X_1| = \ldots = |X_s| = n$.

The problem described above is called *balanced* [16]. Sometimes MAP is formulated in a more general way if $|X_1| = n_1$, $|X_2| = n_2$, $\ldots$, $|X_s| = n_s$ and the requirement $n_1 = n_2 = \ldots = n_s$ is omitted. However, this case can be easily transformed into the balanced problem by complementing the weight matrix to an $n \times n \times \ldots \times n$ matrix with zeros, where $n = \max_i n_i$.

In what follows we assume that the number of dimensions s is a small fixed constant while the size n can be arbitrary large. This corresponds to the real applications (see above) and also follows from the previous research, see, e.g., [10, 88, 93].

1.2.1 Existing Approaches

MAP was studied by many researchers. Several special cases of the problem were intensively studied in the literature (see [70] and references there)

but only for a few classes of them polynomial time exact algorithms were found, see, e.g., [14, 15, 54]. In many cases MAP remains hard to solve [15, 19, 70, 97]. For example, if there are three sets of points of size n on a Euclidean plane and the objective is to find n triples, every triple has a point in each set, such that the total circumference or area of the corresponding triangles is minimal, the corresponding 3-APs are still NP-hard [97]. Apart from proving NP-hardness, researchers studied asymptotic properties of some special instance families [32].

As regards the solution methods, there exist several exact and approximation algorithms [7, 19, 70, 85, 86] and a number of heuristics including construction heuristics [7, 35, 81], greedy randomized adaptive search procedures [1, 77, 81, 93] (including several concurrent implementations, see, e.g., [1, 81]) and a host of local search procedures [1, 7, 8, 15, 16, 52, 81, 93].

Two metaheuristics were proposed for MAP in the literature, namely a simulated annealing procedure [16] and a memetic algorithm [52].

Chapter 2

Some Aspects of Heuristics Design

Heuristic design still mostly depends on the researcher's skills; there are just a few tools to support the scientist in this process. In this chapter we show several examples of how such tools can help in heuristic design.

If a heuristic provides no solution quality or running time guarantee, the only choice to evaluate it is to use empirical analysis. One of the most important aspects of computational experiment design is test bed selection. In Section 2.1 we discuss MAP test bed design.

It turns out that there exist no standard test bed for MAP which would cover at least the most natural cases of the problem. In Section 2.1, we gather all the instance classes proposed in the literature and systematize them. We also split all the instances into two classes according to some important properties. This helps in further experimental analysis.

Unfortunately, there is no way to find the optimal solutions for the instances of the MAP test bed even of a moderate size in any reasonable time. However, in certain circumstances, it is possible to estimate the optimal solution values. In Section 2.2 we show an example of such estimation for one of the most widely used MAP instances family.

In Section 2.3 we show a successful example of producing a test bed for GTSP from a well-known TSP test bed.

Then we show two examples of improvement of heuristic performance. Observe that even a small reduction of the problem size can noticeably speed

up a powerful solver. In Section 2.4 we propose two algorithms intended to reduce the size of a GTSP instance. Our experiments show that this preprocessing may successfully reduce the running time of many GTSP algorithms known from the literature.

Another way to improve heuristic performance is to optimize the algorithm with respect to the hardware architecture. In particular, an extremely important aspect is how the algorithm uses the main memory. Indeed, computer memory is a complicated subsystem and its performance significantly depends on the way it is used. It appears that one has to follow just a few simple rules in order to improve virtually any algorithm to make it 'friendly' with respect to computer memory. In Section 2.5 we use three existing and one new construction heuristics for MAP as an example and show how these algorithms can be improved.

We also discuss the questions of selecting the most convenient and efficient data structures on the example of GTSP in Section 2.6.

2.1 MAP Test Bed

The question of selecting proper test bed is one of the most important questions in heuristic experimental evaluation [89]. While many researchers of MAP focused on instances with random independent weights ([4, 7, 68, 85] and some others) or random instances with predefined solutions [16, 33], several more sophisticated models are of greater practical interest [8, 15, 19, 27, 70]. There is also a number of papers which consider real-world and pseudo real-world instances [10, 77, 84] but we suppose that these instances do not well represent all the instance classes and building a proper benchmark with the real-world instances is a subject for another research.

In this work, we propose splitting all the instance families into two classes: instances with independent weights (Section 2.1.1) and instances with decomposable weights (Section 2.1.2). Later we will show that the heuristics perform differently on the instances of these classes and, thus, this division is very important for experimental analysis.

2.1.1 Instances With Independent Weights

One of the most studied classes of instances for MAP is *Random Instance Family*. In Random, the weight assigned to a vector is a random integer value uniformly distributed in the interval $[a, b-1]$. Random instances were used in [1, 4, 7, 86] and some others. Later, in Section 2.2, we will show that it is possible to estimate the optimal solution value of a large enough Random instance.

Another class of instances with almost independent weights is *GP Instance Family* which contains pseudo-random instances with predefined optimal solutions. GP instances are generated by an algorithm produced by Grundel and Pardalos [33]. The generator is naturally designed for s-AP for arbitrary large values of s and n. However, the generating algorithm is exponential and, thus, it was impossible to generate any GP instances even of a moderate size. Nevertheless, this is what we need since finally we have both small (GP) and large (Random) instances with independent weights with known optimal solutions.

2.1.2 Instances With Decomposable Weights

In many cases it is not easy to define a weight for an s-tuple of objects but it is possible to define a relation between every pair of objects from different sets. In this case one should use *decomposable weights* [98]. Then the weight of a vector e is defined as follows:

$$w(e) = f\left(d^{1,2}_{e_1,e_2}, d^{1,3}_{e_1,e_3}, \ldots, d^{s-1,s}_{e_{s-1},e_s}\right) , \tag{2.1}$$

where $d^{i,j}$ is a weight matrix for the sets X_i and X_j and f is some function.

The most straightforward instance family with decomposable weights is Clique. It defines the function f as the sum of all the arguments:

$$w_{\mathrm{c}}(e) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} d^{i,j}_{e_i,e_j} . \tag{2.2}$$

The Clique instance family was investigated in [8, 19, 27] and some others. It was proven [19] that MAP restricted to Clique instances remains NP-hard.

A special case of Clique is *Geometric Instance Family*. In Geometric, each of the sets $X_1, X_2, \ldots, X_s$ corresponds to a set of points in Euclidean space, and the distance between two points $u \in X_i$ and $v \in X_j$ is defined as Euclidean distance; we consider the two dimensional Euclidean space:

$$d_{\mathrm{g}}(u, v) = \sqrt{(u_x - v_x)^2 + (u_y - v_y)^2}\,.$$

It is proven [97] that the Geometric instances are NP-hard to solve for $s = 3$ and, thus, Geometric is NP-hard for every $s \geq 3$.

We propose a new instance family with decomposable weights, SquareRoot. It is a modification of the Clique instance family. Assume we have s radars and n planes and each radar observes all the planes. The problem is to assign signals which come from different radars to each other. It is quite natural to define some distance function between each pair of signals from different radars which would correspond to the similarity of these signals. Then for a set of signals which correspond to one plane the sum of these distances is expected be small and, hence, (2.2) is a good choice. However, it is not actually correct to minimize the total distance between the signals; one should also ensure that none of these distances is too large. Note that the same requirement appears in a number of other applications. We propose a weight function which aims to both small total distance between the assigned signals and small dispersion of these distances:

$$w_{\mathrm{sq}}(e) = \sqrt{\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \left(d^{i,j}_{e_i,e_j}\right)^2}\,. \tag{2.3}$$

Similar approach is used in [70] though they do not use square root, i.e., a vector weight is just a sum of squares of the edge weights in a clique. In addition, the edge weights in [70] are calculated as distances between some nodes in a Euclidean space.

Another special case of the decomposable weights, Product, is studied in [15]. Burkard et al. consider 3-AP and define the weight $w(e)$ as $w(e) = a^1_{e_1} \cdot a^2_{e_2} \cdot a^3_{e_3}$, where a^1, a^2 and a^3 are random vectors of positive numbers. We generalize the Product instance family for s-AP: $w_{\mathrm{p}}(e) = \prod_{i=1}^{s} a^i_{e_i}$. It is easy to

show that the Product weight function can be represented in the form (2.1). Note that the minimization problem for the Product instances is proven to be NP-hard in case $s = 3$ and, thus, it is NP-hard for every $s \geq 3$.

2.1.3 Additional Details

We include the following instances in our test bed:

- Random instances where each weight was randomly chosen in $\{1, 2, \ldots, 100\}$, i.e., $a = 1$ and $b = 101$. We will show in Section 2.1.1 that the weights of the optimal solutions of all the considered Random instances are very likely to be $an = n$.
- GP instances with predefined optimal solutions.
- Clique and SquareRoot instances, where the weight of each edge in the graph was randomly selected from $\{1, 2, \ldots, 100\}$. Instead of the optimal solution value we use the best known solution value.
- Geometric instances, where both coordinates of every point were randomly selected from $\{1, 2, \ldots, 100\}$. The distances between the points are calculated precisely while the weight of a vector is rounded to the nearest integer. Instead of an optimal solution value we use the best known solution value.
- Product instances, where every value a_i^j was randomly selected from $\{1, 2, \ldots, 10\}$. Instead of an optimal solution value we use the best known solution value.

An instance name consists of three parts: the number s of dimensions, the type of the instance ('gp' for GP, 'r' for Random, 'cq' for Clique, 'g' for Geometric, 'p' for Product and 'sr' for SquareRoot), and the size n of the instance. For example, `5r40` means a five dimensional Random instance of size 40. For every combination of instance size and type we generated 10 instances, using the number $seed = s+n+i$ as a seed of the random number sequences, where i is an index of the instance of this type and size, $i \in \{1, 2, \ldots, 10\}$. Thereby, every experiment is conducted for 10 different instances of some fixed type

and size, i.e., every number reported in the tables below is an average for 10 runs, one for each of the 10 instances. This smooths out the experimental results.

2.2 Probabilistic Analysis of Test Beds

In experimental analysis of heuristics, it is important to know the optimal solutions for test bed instances. However, it is not always possible to solve every instance to optimality. Indeed, heuristic approach is usually applied when exact algorithms fail to solve the problem in any reasonable time.

If the optimal solution cannot be obtained for an instance, one can use a lower or an upper bound instead or apply probabilistic analysis of the instance if it is generated randomly. In this section we apply the latter approach to the MAP Random instance family.

Recall that in Random instances, the weight assigned to a vector is an independent random uniformly distributed integer in the interval $[a, b-1]$. Let us estimate the average solution value for Random. In fact, we prove that it is very likely that every large enough Random instance has an assignment of weight an, i.e., a minimal possible assignment (observe that a minimal assignment includes n vectors of weight a).

Let α be the number of assignments of weight na and let $c = b - a$. We would like to have an upper bound on the probability $\Pr(\alpha = 0)$. Such an upper bound is given in the following theorem whose proof is based on the Extended Jansen Inequality given in Theorem 8.1.2 of [2].

Theorem 1 *For values of n such that $n \geq 3$ and*

$$\left(\frac{n-1}{e}\right)^{s-1} \geq c \cdot 2^{\frac{1}{n-1}}, \tag{2.4}$$

we have $\Pr(\alpha = 0) \leq e^{-\frac{1}{2\sigma}}$, where $\sigma = \sum_{k=1}^{n-2} \frac{\binom{n}{k} \cdot c^k}{[n \cdot (n-1) \cdots (n-k+1)]^{s-1}}$.

Proof. Let $[n] = \{1, 2, \ldots, n\}$. Let t be the number of all feasible assignments and let A be an arbitrary assignment consisting of vectors $e^1, e^2, \ldots, e^n$

such that $e_1^i = i$ for each $i \in [n]$. There are $n!$ possibilities to choose the jth coordinate of all vectors in A for each $j = 2, 3, \ldots, n$ and, thus, $t = (n!)^{s-1}$.

Let R be the set of vectors in X of weight a and let $\{A_1, A_2, \ldots, A_t\}$ be the set of all assignments. Let B_i be the event $\{A_i \subset R\}$ for each $i \in [t]$. Let $\mu = \sum_{i=1}^{t} \Pr(B_i)$ and $\Delta = \sum_{i \sim j} \Pr(B_i \cap B_j)$, where $i \sim j$ if $i \neq j$ and $A_i \cap A_j \neq \varnothing$ and the sum for Δ is taken over all ordered pairs (B_i, B_j) with $i \sim j$.

By the Extended Jansen Inequality,

$$\Pr(\alpha = 0) \leq e^{-\frac{\mu^2}{2\Delta}} \tag{2.5}$$

provided $\Delta \geq \mu$. We will compute μ and estimate Δ to apply (2.5) and to show $\Delta \geq \mu$. It is easy to see that $\mu = \frac{t}{c^n}$.

Now we will estimate Δ. Let $A_i \cap A_j = K$, $k = |K|$ and $i \neq j$. Thus, we have

$$\Pr(B_i \cap B_j) = \Pr(K \subset R) \cdot \Pr(A_i \backslash K \subset R) \cdot \Pr(A_j \backslash K \subset R) = \frac{1}{c^k} \left(\frac{1}{c^{n-k}} \right)^2 = \frac{1}{c^{2n-k}}.$$

Let $(f^1, f^2, \ldots, f^n)$ be an assignment with $f_1^i = i$ for every $i \in [n]$ and consider the following two sets of assignments. Let

$$P(k) = \{(e^1, e^2, \ldots, e^n) : \ \forall i \in [n] \ (e_1^i = i) \text{ and } \forall j \in [k] \ (e^j = f^j)\}$$

and let $Q(n-k) = \{(e^1, e^2, \ldots, e^n) : \ \forall i \in [n] \ (e_1^i = i) \text{ and } \forall j \in [n-k] \ (e^{k+j} \neq f^{k+j})\}$. Let $h(n,k) = |P(k) \cap Q(n-k)|$. Clearly, $h(n,k) \leq |P(k)| = \big((n-k)!\big)^{s-1}$. Observe that

$$h(n,k) \geq |P(k)| - (n-k)|P(k+1)| = L(n,k,s),$$

where $L(n,k,s) = \big((n-k)!\big)^{s-1} - (n-k) \cdot \big((n-k-1)!\big)^{s-1}$.

Let $g(n,k)$ be the number of ordered pairs (A_i, A_j) such that $|A_i \cap A_j| = k$. Observe that $g(n,k) = t \cdot \binom{n}{k} \cdot h(n,k)$ and, thus, $t \cdot \binom{n}{k} \cdot L(n,k,s) \leq g(n,k) \leq t \cdot \binom{n}{k} \cdot \big((n-k)!\big)^{s-1}$.

Observe that $\Delta = \sum_{k=1}^{n-2} \sum_{|A_i \cap A_j|=k} \Pr(B_i \cap B_j) = \sum_{k=1}^{n-2} g(n,k) \cdot c^{k-2n}$. Thus,

$$\frac{(n!)^{s-1}}{c^{2n}} \cdot \sum_{k=1}^{n-2} \binom{n}{k} \cdot c^k \cdot L(n,k,s) \leq \Delta \leq \frac{(n!)^{s-1}}{c^{2n}} \sum_{k=1}^{n-2} \binom{n}{k} \cdot c^k \cdot \big((n-k)!\big)^{s-1} \quad (2.6)$$

Now $\Pr(\alpha = 0) \leq e^{-\frac{1}{2\sigma}}$ follows from (2.5) by substituting μ with $\frac{(n!)^{s-1}}{c^n}$ and Δ with its upper bound in (2.6). It remains to prove that $\Delta \geq \mu$. Since $n \geq 3$, $L(n,1,s) \geq \frac{1}{2}\big((n-1)!\big)^{s-1}$. By the lower bound for Δ in (2.6), we have $\Delta \geq \frac{(n!)^{s-1}}{c^{2n-1}} \cdot L(n,1,k)$. Therefore, $\frac{\Delta}{\mu} \geq \frac{0.5((n-1)!)^{s-1}}{c^{n-1}}$. Now using the inequality $(n-1)! > (\frac{n-1}{e})^{n-1}$, we conclude that $\frac{\Delta}{\mu} \geq 1$ provided (2.4) holds. □

Useful results can also be obtained from (11) in [32] that is an upper bound for the average optimal solution. Grundel, Oliveira and Pardalos [32] consider the same instance family except the weights of the vectors are real numbers uniformly distributed in the interval $[a,b]$. However the results from [32] can be extended to our discrete case. Let $w'(e)$ be a real weight of the vector e in a continuous instance. Consider a discrete instance with $w(e) = \lfloor w'(e) \rfloor$ (if $w'(e) = b$, set $w(e) = b-1$). Note that the weight $w(e)$ is a uniformly distributed integer in the interval $[a, b-1]$. The optimal assignment weight of this instance is not larger than the optimal assignment weight of the continuous instance and, thus, the upper bound for the average optimal solution for the discrete case is correct.

In fact, the upper bound $\bar{z}_u^*$ (see [32]) for the average optimal solution is not really accurate. For example, $\bar{z}_u^* \approx an + 6.9$ for $s = 3$, $n = 100$ and $b - a = 100$, and $\bar{z}_u^* \approx an + 3.6$ for $s = 3$, $n = 200$ and $b - a = 100$. It gives a better approximation for larger values of s, e.g., $\bar{z}_u^* \approx an + 1.0$ for $s = 4$, $n = 40$ and $b - a = 100$, but Theorem 1 provides stronger results ($\Pr(\alpha > 0) \approx 1.000$ in the latter case).

The following table gives the probabilities for $\Pr(\alpha > 0)$ for various values of s and n:

$s = 4$		$s = 5$		$s = 6$		$s = 7$	
n	$\Pr(\alpha > 0)$	n	$\Pr(\alpha > 0)$	n	$\Pr(\alpha > 0)$	n	$\Pr(\alpha > 0)$
15	0.575	10	0.991	8	1.000	7	1.000
20	0.823	11	0.998				
25	0.943	12	1.000				
30	0.986						
35	0.997						
40	1.000						

2.3 GTSP Test Bed

There is a standard test bed which was used in most of the recent literature on GTSP, see, e.g., [95, 96, 101, 100]. It is produced from a well-known TSP test bed called TSPLIB [91]. TSPLIB contains both symmetric and asymmetric instances of different sizes and types, some pseudo-random and some real-world ones.

The procedure of generating a GTSP instance from a TSP instance was proposed by Fischetti, Salazar, and Toth [25]. It is applicable to both symmetric and asymmetric TSP instances and produces symmetric and asymmetric GTSP instances, respectively. The number of vertices n in the produced GTSP instance is the same as in the original TSP instance; the number m of clusters is $m = \lceil n/5 \rceil$. The clusters are 'localized', i.e., the procedure attempts to group close vertices into the same clusters.

TSPLIB includes instances from as few as 14 to as many as 85900 vertices. In our experiments, we usually consider instances with $10 \leq m \leq 217$. The same test bed is used in a number of papers, see, e.g., [12, 41, 58, 95]. In other papers the bounds are even smaller.

Every instance name consists of three parts: 'm t n', where m is the number of clusters, t is the type of the original TSP instance (see [91] for details) and n is the number of vertices.

Observe that the optimal solutions are known only for certain instances with up to 89 clusters [25]. For the rest of the test bed we use the best known solutions obtained in our experiments and from the literature [12, 95].

2.4 Preprocessing

Preprocessing is a procedure of an instance simplification. It is used to reduce the computation time of a solver. There are several examples of such approaches in integer and linear programming (e.g., [44, 94]) as well as for the Vehicle Routing Problem [72]. In some cases preprocessing plays the key role in an algorithm (see, e.g., [26]). Next we propose two efficient preprocessing procedures for GTSP.

2.4.1 GTSP Reduction Algorithms

An important feature of GTSP is that a feasible tour does not visit every vertex of the problem and, thus, GTSP may contain vertices that a priori cannot be included in the optimal tour and, hence, may be removed in advance.

Definition 1 *Let C be a cluster, $|C| > 1$. We say that a vertex $r \in C$ is* redundant *if, for each pair x and y of vertices from distinct clusters different from C, there exists $r' \in C \setminus \{r\}$ such that $w(x \to r' \to y) \leq w(x \to r \to y)$.*

Testing this condition for every vertex takes approximately $O(n^3 s)$ operations. In some cases it is possible to significantly reduce the preprocessing time for symmetric instances.

Let us fix some vertex $r \in C$. For every $r' \in C$ and every $x \notin C$ calculate the value $\Delta_x^{r,r'} = w(x \to r) - w(x \to r')$. Observe that a vertex r is redundant if there is no pair of vertices $x, y \notin C$ from different clusters such that $\Delta_x^{r,r'} + \Delta_y^{r,r'} < 0$ for every r', i.e., r is redundant if for every $x, y \notin C$, $\mathit{Cluster}(x) \neq \mathit{Cluster}(y)$, there exists $r' \in C \setminus \{r\}$ such that $\Delta_x^{r,r'} + \Delta_y^{r,r'} \geq 0$. That is due to $\Delta_x^{r,r'} + \Delta_y^{r,r'} = w(x \to r) - w(x \to r') + w(y \to r) - w(y \to r') = w(x \to r \to y) - w(x \to r' \to y)$.

There is a way to accelerate the algorithm. If

$$\min_{x \notin Z} \max_{r' \in C} \Delta_x^{r,r'} + \min_{x \in Z} \max_{r' \in C} \Delta_x^{r,r'} < 0$$

for some cluster Z, then we know immediately that r cannot be reduced. We can use an equivalent condition:

$$\min_{x \in \bigcup_{j<i} V_j} \max_{r' \in C} \Delta_x^{r,r'} + \min_{x \in V_i} \max_{r' \in C} \Delta_x^{r,r'} < 0$$

This condition can be tested during the Δ values calculation by accumulating the value of $\min_{x \in \bigcup_{j<i} V_j} \max_{r' \in C} \Delta_x^{r,r'}$.

Removing a redundant vertex may cause a previously irredundant vertex to become redundant. Thus, it is useful to check redundancy of vertices in cyclic order until we see that, in the last cycle, no vertices are found to be redundant. However, in the worst case, that would lead to the total number of the redundancy tests to be $\Theta(n^2)$. Our computational experience has shown that almost all redundant vertices are found in two cycles. Hence, we never conduct the redundancy test more than twice for a vertex.

Similar to the vertices, it is sometimes possible to say in advance that a certain edge cannot be included in the optimal tour.

Definition 2 *Let u and v be a pair of vertices from distinct clusters U and C respectively. Then the edge $u \to v$ is* redundant *if for each vertex $x \in V \setminus (U \cup C)$ there exists $v' \in C \setminus \{v\}$ such that $w(u \to v' \to x) \leq w(u \to v \to x)$.*

We propose the following algorithm for the edge reduction. Given a vertex $v \in C$, $|C| > 1$, we detect redundant edges incident with v using the following procedure:

1. Select an arbitrary vertex $v'' \in C \setminus \{v\}$.
2. Set $P_x = \Delta_x^{v,v''}$ for each vertex $x \in V \setminus C$.
3. Sort the array P in non-decreasing order.
4. For each cluster $U \neq C$ and for each vertex $u \in U$ do the following:
   (a) $\delta = \Delta_u^{v,v''}$.
   (b) For each item $\Delta_x^{v,v''}$ of the array P such that $\Delta_x^{v,v''} + \delta < 0$ check: if $x \notin U$ and $\Delta_x^{v,v'} + \Delta_u^{v,v'} < 0$ for every $v' \in C \setminus \{v, v''\}$, the edge $u \to v$ is immediately not redundant, continue with the next u.

(c) Edge $u \to v$ is redundant, set $w(u \to v) = \infty$.

To prove that the above edge reduction algorithm works correctly, let us fix some edge $u \to v$, $u \in U$, $v \in C$, $U \neq C$. The algorithm declares this edge redundant if the following condition holds for each $x \notin C$ (see 4b):

$$\begin{aligned} \Delta_x^{v,v''} + \Delta_u^{v,v''} \geq 0 \qquad & \text{or} \\ \Delta_x^{v,v'} + \Delta_u^{v,v'} \geq 0 \qquad & \text{for some } v' \in C \setminus \{v, v''\}\,. \end{aligned}$$

This is equivalent to

$$\Delta_x^{v,v'} + \Delta_u^{v,v'} \geq 0 \qquad \text{for some } v' \in C \setminus \{v\}\,.$$

So the algorithm declares the edge $u \to v$ redundant if for each $x \in V \setminus C \setminus U$ there exists $v' \in C \setminus \{v\}$ such that $\Delta_x^{v,v'} + \Delta_u^{v,v'} \geq 0$:

$$\begin{aligned} & w(x \to v) - w(x, v') + w(u \to v) - w(u \to v') \geq 0 \text{ and, hence,} \\ & w(u \to v \to x) \geq w(u \to v' \to x)\,. \end{aligned}$$

The edge reduction procedure is executed exactly once for every vertex v such that $|\mathit{Cluster}(v)| > 1$. The whole algorithm takes $O(n^3 s)$ operations.

2.4.2 GTSP Reduction Experimental Evaluation

We have tested three reduction algorithms: the Vertex Reduction Algorithm, the Edge Reduction Algorithm, and the Combined Algorithm which first applies the Vertex Reduction and then the Edge Reduction.

The columns of Table A.5 are as follows:

- *Instance* is the instance name. Note that the suffix number in the name is the number of vertices before any preprocessing.
- R_v is the number of vertices detected as redundant.
- R_e is the number of edges detected as redundant. For the Combined Algorithm, R_e shows the number of redundant edges in the instances already reduced by the Vertex Reduction.

- T is preprocessing time, in seconds.

All the algorithms are implemented in C++; the evaluation platform is based on an AMD Athlon 64 X2 Core Dual processor (3 GHz frequency).

The results of the experiments (Table A.5) show that the preprocessing time for the Vertex Reduction is negligible (less than 50 ms) for all the instances up to `212u1060`, i.e., for almost all `TSPLIB`-based GTSP instances used in the literature. The average percentage of detected redundant vertices for these instances is 14%, and it is 11% for all considered instances. The experimental complexity of the Vertex Reduction algorithm is about $O(n^{2.4})$.

The Edge Reduction is more time-consuming than the Vertex Reduction. The running time is negligible for all instances up to `115rat575`. Note that in most of the GTSP literature, only instances with $m \leq 89$ are considered. The average per cent of the detected redundant edges for these instances is about 27%, and it is 21% for all the instances in Table A.5. The experimental complexity of the Edge Reduction algorithm is $O(n^{2.6})$.

2.4.3 Influence of GTSP Reduction on Solvers

Certainly, one can doubt the usefulness of our reduction algorithms since they may not necessarily decrease the running time of GTSP solvers. Therefore, we have experimentally checked if the reductions are beneficial for several powerful GTSP solvers (obviously, preprocessing is useless in combination with a fast solver since preprocessing may take more time than the solver itself):

- An exact algorithm (Exact) based on a transformation of GTSP to TSP [11]; the algorithm from [23] was not available. The algorithm that we use converts a GTSP instance with n vertices to a TSP instance with $3n$ vertices in the polynomial time, solves the obtained TSP using the Concorde solver [5], and then converts the obtained TSP solution to GTSP solution also in the polynomial time.

- A memetic algorithm from [96] (SD).

- A memetic algorithm from [95] (SG).

- A modified version of our memetic algorithm (see Section 5.1) (GK).

For each stochastic algorithm, every test was repeated ten times and an average was used. The columns of the tables not described above are as follows:

- T_0 is the original solution time.
- B is the time benefit, i.e., $\frac{T_0 - T_{\text{pr}}}{T_0} \cdot 100\%$, where T_{pr} is the solution time after preprocessed; this includes the preprocessing time.

The experiments (see Tables A.1, A.2, A.3 and A.4) show that the Vertex Reduction, the Edge Reduction and the Combined Reduction Techniques significantly reduce the running time of the Exact and SD solvers. However, the Edge Reduction (and because of that the Combined Reduction Technique) is not that successful for SG (Table A.3) and the original version of GK. That is because not every algorithm processes infinite or enormous edges well.

We have adjusted our solver GK to work better with preprocessed instances. The details of the modified version can be found in Section 5.1.10. The modified algorithm does not reproduce exactly the results of the original GK heuristic; it produces slightly better solutions at the cost of slightly larger running times. However, one can see (Table A.4) that all the Reduction Algorithms proposed in this section influence the modified GK algorithm positively.

Different reductions have different degree of success for different solvers. The Edge Reduction is more efficient than the Vertex Reduction for SD; in other cases the Vertex Reduction is more successful. For every solver except SG the Combined Technique is preferred to single reductions.

Preprocessing is called to reduce the solution time. On the other hand, there is no guarantee that the outcome of the preprocessing will be noticeable. Thus, it is important to ensure that, at least, preprocessing is significantly faster than the solver.

Four GTSP solvers are considered in this section. The first solver, Exact, is exponential and, thus, it is clear that its time complexity is larger than the one of the reduction algorithms. The time complexities of the other three

solvers were estimated experimentally. The experimental complexity of SD is about $\Theta(n^3)$ and it is about $\Theta(n^{3.5})$ for GK and SG.

Since the time complexity of every considered solver is higher than the time complexity of preprocessing, we can conclude that preprocessing remains relatively fast for an arbitrary large instance.

Note that the solution quality of the considered solvers was not affected by the reductions, on average.

2.4.4 MAP Preprocessing

We do not discuss any preprocessing for MAP. Observe that the Vertex Reduction proposed above for GTSP preserves the structure of the problem and, hence, any ordinary GTSP solver may be applied to a reduced instance. However, it is not the case for the Edge Reduction (see above) since the yielded instances may contain infinite edges.

It is unlikely that a MAP preprocessing can reduce the values of s or n. One can rather think of removing certain vectors from the X set, however, this would change the problem structure and, hence, complicate the solvers.

2.5 Implementation Performance Issues

It may seem that implementation is a technical question which is not worth discussion because its influence on the algorithm's performance is negligible and, moreover, platform-dependent. In this section we will show that this assumption is sometimes very wrong. It turns out that two formally equal implementations of some algorithm may have very different running times in certain circumstances. In particular, we will show that some simple transformations of an algorithm may be crucial with respect to efficiency of processor cache usage. Note that this discussion does not involve any specifics of particular CPUs and is relevant to all the computers produced in at least last 30 years.

For a case study we need some algorithms which deal with large amounts of data. For this purpose we selected MAP construction heuristics. Recall that MAP instance is defined by an s-dimensional matrix of size n, i.e., it

requires n^s values. Construction heuristics are very quick and, hence, we are able to consider very large instances such that the weight matrices exceed the size of processor cache.

In Section 2.5.1 we describe all MAP construction heuristics known from the literature and propose a new one, Shift-ROM. In Section 2.5.2 we discuss the efficiency of computer memory in certain circumstances and provide several simple rules to improve performance of an algorithm implementation. Then, in Section 2.5.3, we show how these rules can be applied to the MAP construction heuristics. We do not provide any experimental analysis here; one can refer to [62] for details. We only declare here that the refinements proposed below speed up each of the considered heuristics in, roughly speaking, 2 to 5 times.

2.5.1 MAP Construction Heuristics

Greedy

The Greedy heuristic starts with an empty partial assignment $A = \varnothing$. On each of n iterations Greedy finds a vector $e \in X$ of minimum weight, such that $A \cup \{e\}$ is a feasible partial assignment, and adds it to A.

The time complexity of Greedy heuristic is $O(n^s + (n-1)^s + \ldots + 2^s + 1) = O(n^{s+1})$ (if the Greedy algorithm is implemented via sorting of all the vectors according to their weights, the algorithm complexity is $O(n^s \cdot \log n^s)$ however this implementation is inefficient, see Section 2.5.3).

Max-Regret

The Max-Regret heuristic was first introduced in [7] for 3-AP and its modifications for s-AP were considered in [10].

Max-Regret proceeds as follows. Initialize a partial assignment $A = \varnothing$. Set $V_d = \{1, 2, \ldots, n\}$ for each $1 \le d \le s$. For each dimension d and each coordinate value $v \in V_d$ consider every vector $e \in X'$ such that $e_d = v$, where $X' \subset X$ is the set of 'available' vectors, i.e., $A \cup \{e\}$ is a feasible partial assignment if and only if $e \in X'$. Find two vectors ${e^1}_{\min}$ and ${e^2}_{\min}$ in the considered subset $Y_{d,v} = \{e \in X' : \ e_d = v\}$ such that ${e^1}_{\min} = \operatorname*{argmin}_{e \in Y_{d,v}} w(e)$,

and ${e^2}_{\min} = \underset{e \in Y_{d,v} \setminus \{{e^1}_{\min}\}}{\operatorname{argmin}} w(e)$. Select the pair (d, v) that corresponds to the maximum difference $w({e^2}_{\min}) - w({e^1}_{\min})$ and add the vector ${e^1}_{\min}$ for the selected (d, v) to A.

The time complexity of Max-Regret is $O(s \cdot n^s + s \cdot (n-1)^s + \ldots + s \cdot 2^s + s) = O(s \cdot n^{s+1})$.

ROM

The *Recursive Opt Matching* (ROM) is introduced in [35] as a heuristic of large domination number (see [35] for definitions and results in domination analysis). ROM proceeds as follows. Initialize A with a trivial assignment: $A^i = (i, i, \ldots, i)$. On each jth iteration of the heuristic, $j = 1, 2, \ldots, s-1$, calculate an $n \times n$ matrix $M_{i,v} = \sum_{e \in Y(j,i,v)} w(e)$, where $Y(j, i, v)$ is a set of all vectors $e \in X$ such that the first j coordinates of the vector e are equal to the first j coordinates of the vector A^i and the $(j+1)$th coordinate of e is v: $Y(j, i, v) = \{e \in X : \ e_k = A^i_k, 1 \le k \le j \text{ and } e_{j+1} = v\}$. Let permutation π be a solution of the 2-AP for the matrix M. Set $A^i_{j+1} = \pi(i)$ for each $1 \le i \le n$.

The time complexity of ROM heuristic is $O((n^s + n^3) + (n^{s-1} + n^3) + \ldots + (n^2 + n^3)) = O(n^s + sn^3)$.

Shift-ROM

A disadvantage of the ROM heuristic is that it is not symmetric with respect to the dimensions. For example, if the vector weights do not depend significantly on the last coordinate then the algorithm is likely to work badly. Shift-ROM is intended to solve this problem by trying ROM for different permutations of the instance dimensions. However, we do not wish to try all $s!$ possible dimension permutations as that would increase the running time of the algorithm quite significantly. We apply only s permutations: $(X_1 X_2 \ldots X_s)$, $(X_s X_1 X_2 \ldots X_{s-1})$, $(X_{s-1} X_s X_1 X_2 \ldots X_{s-2})$, $\ldots$, $(X_2 X_3 \ldots X_s X_1)$.

In other words, on each run Shift-ROM applies ROM to the problem; upon completion, it renumbers the dimensions for the next run in the following way: $X_1 \leftarrow X_2$, $X_2 \leftarrow X_3$, $\ldots$, $X_{s-1} \leftarrow X_s$, $X_s \leftarrow X_1$. After s runs, the

best solution is selected.

The time complexity of Shift-ROM heuristic is $O((n^s + sn^3) \cdot s) = O(sn^s + s^2n^3)$.

Time Complexity Comparison

Now we can gather all the information about the time complexity of the considered heuristics. The following table shows the time complexity of each of the heuristics for different values of s:

	Greedy	Max-Regret	ROM	Shift-ROM
Arbitrary s	$O(n^{s+1})$	$O(sn^{s+1})$	$O(n^s + sn^3)$	$O(sn^s + s^2n^3)$
Fixed $s = 3$	$O(n^4)$	$O(n^4)$	$O(n^3)$	$O(n^3)$
Fixed $s \geq 4$	$O(n^{s+1})$	$O(n^{s+1})$	$O(n^s)$	$O(n^s)$

2.5.2 Performance Notes

In a standard computer model it is assumed that all the operations take approximately the same time. However, it is not true since the architecture of a modern computer is complex. We will use a more sophisticated model in our further discussion. The idea is to differentiate fast and slow memory access operations.

The weight matrix of a MAP instance is normally stored in the Random Access Memory (RAM) of a computer. RAM's capacity is large enough even for very large instances, e.g., nowadays RAM of an average desktop PC is able to hold a weight matrix for 3-AP with $n = 750$, i.e., $4.2 \cdot 10^8$ weights[1]. RAM is a fast storage; one can load gigabytes of data from RAM in one second. However, RAM has a comparatively high latency, i.e., it takes a lot of time for the processor to access even a small portion of data in RAM. Processor cache is intended to minimize the time spent by the processor for waiting for RAM response.

The processor cache exploits two heuristics: firstly, if some data was recently used then there is a high probability that it will be used again soon,

[1]Here and further we assume that every weight is represented with a 4 byte integer. The calculations are provided for 2 Gb of RAM.

and, secondly, the data is usually used successively, i.e., if some portion of data is used now then it is likely that the successive portion of data will be used soon. As an example, consider an in place vector multiplication algorithm: on every iteration the algorithm loads a value from the memory, multiplies it and saves the result at the same memory position. So, the algorithm accesses every portion of data twice and the data is accessed successively, i.e., the algorithm accesses the first element of the vector, then it accesses the second element, the third one, etc.

Processor cache[2] is a temporary data storage, relatively small and fast, usually located on the same chip as the processor. It contains several *cache lines* of the same size; each cache line holds a copy of some fragment of the data stored in RAM. Each time the processor needs to access some data in RAM it checks whether this data is already presented in the cache. If this is the case, it accesses this data in the cache instead. Otherwise that if a 'miss' is detected, the processor suspends, some cache line is freed and a new portion of data is loaded from RAM to cache. Then the processor resumes and accesses the data in the cache as normally. Note that in case of a 'miss' the system loads the whole cache line that is currently 64 bytes on most of the modern computers [3] and this size tends to grow with the development of computer architecture. Thus, if a program accesses some value in the memory several times in a short period of time it is very likely that this data will be loaded from RAM just once and then will be stored in the cache so the access time will be minimal. Moreover, if some value is accessed and, thus, loaded from RAM to the processor cache, it is likely that the next value is also loaded since the cache line is large enough to store several values.

With respect to MAP heuristics, there are two key rules for improving the memory subsystem performance:

1. The successive access to the weight matrix (scan), i.e., access to the matrix in the order of its alignment in the memory, is strongly preferred (we use the row-major order [64] for weight matrix in our implementations of the algorithms). Note that if an algorithm accesses, e.g., every second weight in the matrix and does it in the proper order, the real complexity of this scan with respect to the memory subsystem is the

[2]We provide a simplified overview of cache; for detailed information, see, e.g., [6].

same as the complexity of a full scan since loading of one value causes loading of several neighbor values.

2. One should minimize the number of the weight matrix scans as much as possible. Even a partial matrix scan is likely to access much more data than the processor cache is able to store, i.e., the data will be loaded from RAM all over again for every scan.

Following these rules may significantly improve the running time of the heuristics. In our experiments, the benefit of following these rules was a speed-up of roughly speaking 2 to 5 times.

2.5.3 MAP Construction Heuristics Improvement

Greedy Heuristic Optimization

A common implementation of the greedy approach for a combinatorial optimization problem involves sorting of all the weights in the problem. In case of MAP this approach is inefficient since we actually need only n vectors from the set of size n^s. Another natural implementation of the Greedy heuristic is to scan all available vectors and to choose the lightest one on each iteration but it is very unfriendly with respect to the memory subsystem: it performs n scans of the weight matrix.

We propose a combination of these approaches; our algorithm proceeds as follows. Let $A = \varnothing$ be a partial assignment and B an array of vectors. While $|A| < n$, i.e., A is not a full assignment, the following is repeated. We scan the weight matrix to fill the array B with k vectors corresponding to k minimal weights in non-decreasing order: if the weight of the current vector is less than the largest weight in B then we insert the current vector to B in the appropriate position and, if necessary, remove the last element of B. Then, for each vector $e \in B$, starting from the lightest, we check whether $A \cup \{e\}$ is a feasible partial assignment and, if so, add e to A. Note, that during the second and further cycles we scan not the whole weight matrix but only a subset $X' \subset X$ of the vectors that can be included into the partial assignment A with the feasibility preservation: $A \cup \{x\}$ is a partial assignment

for any $x \in X'$. The size of the array B is calculated as $k = \min\{64, |X'|\}$ in our implementation. The constant 64 is obtained empirically.

The algorithm is especially efficient on the first iterations, i.e., in the hardest part of its work, while the most of the vectors are feasible. However, there exists a bad case for this heuristic. Assume that the weight matrix contains a lot of vectors of the minimal weight $w_{\min}$. Then the array B will be filled with vectors of the weight $w_{\min}$ at the beginning of the scan and, thus, it will contain a lot of similar vectors (recall that the weight matrix is stored in the row-major order and only the last coordinates are varied at the beginning of the scan, so all the vectors processed at the beginning of the scan are likely to have the same first coordinates). As a result, selecting the first of these vectors will cause infeasibility for the other vectors in B. We use an additional heuristic to decrease the running time of the Greedy algorithm for such instances. Let $w_{\min}$ be the minimum possible weight: $w_{\min} = \min_{e \in X'} w(e)$ (sometimes this value is known like for Random instance family it is 1, see Section 2.1). If it occurs during the matrix scan that all the vectors in B have the weight $w_{\min}$, i.e., $w(B_i) = w_{\min}$ for every $1 \leq i \leq k$, then the rest of the scan can be skipped because there is certainly no vector lighter than B_k. Moreover, it is safe to update $w_{\min}$ with $w(B_k)$ every time before the next matrix scan.

Max-Regret Heuristic Optimization

The Max-Regret heuristic naturally requires $O(n^2 s)$ weight matrix partial scans. Each of these scans fixes one coordinate and, thus, every available vector $e \in X'$ (see Subsection 2.5.3) is accessed s times during each iteration, and this access is very inefficient when the last coordinate is fixed (recall that the weight matrix is stored in a row-major order and, thus, if the last coordinate is fixed then the algorithm accesses every nth value in the memory, i.e., the access is very non-successive and one can assume that this scan will load the whole weight matrix from RAM to cache). In our more detailed computer model (see Section 2.5.2), the time complexity of the non-optimized Max-Regret is $O((s-1) \cdot n^{s+1} + n^{s+2})$.

We propose another way to implement Max-Regret. Let us scan the whole set X' of available vectors on each iteration. Let L be an $n \times s$ matrix of

the lightest vector pairs: $L^1_{i,j}$ and $L^2_{i,j}$ are the lightest vectors when the jth coordinate is fixed as i, and $w(L^1_{i,j}) \leq w(L^2_{i,j})$. To fill the matrix L we do the following: for every vector $e \in X'$ and for every coordinate $1 \leq d \leq s$ check: if $w(e) < w(L^1_{e_d,d})$, set $L^2_{e_d,d} = L^1_{e_d,d}$ and $L^1_{e_d,d} = e$. Otherwise if $w(e) < L^2_{e_d,d}$, set $L^2_{e_d,d} = e$. Thus, we update the $L_{e_d,d}$ item of the matrix with the current e if $w(e)$ is small enough. Having the matrix L, we can easily find the coordinate d and the fixed value v such that $w(L^2_{v,d}) - w(L^1_{v,d})$ is maximized. The vector $L^1_{v,d}$ is added to the solution and the next iteration of the algorithm is executed.

The proposed algorithm performs just n partial scans of the weight matrix. The matrix L is usually small enough to fit in the processor cache, so the access to L is fast. Thus, the time complexity of the optimized Max-Regret in our more detailed computer model is $O(n^{s+1})$.

ROM Heuristic Optimization

The ROM heuristic can be implemented in a very friendly way with respect to the memory access. On the first iteration it fixes the first two coordinates (n^2 combinations) and enumerates all vectors with these fixed coordinates. Thus, it scans the whole weight matrix successively. On the next iteration it fixes three coordinates (n^2 combinations as the second coordinate depends on the first one), and enumerates all vectors with these fixed coordinates. Thus, it scans n^2 solid n^{s-3}-size fragments of the weight matrix; further iterations are similar. As a result, the time complexity of ROM in our more detailed computer model is the same as in a simple one: $O(n^s + sn^3)$.

Shift-ROM Heuristic Optimization

The Shift-ROM heuristic is an extension of ROM; it simply runs ROM s times, starting it from different dimensions. However, not every run of ROM is efficient when it is a part of Shift-ROM. Let us consider the case when the first iteration of ROM fixes the last two coordinates. For each of the n^2 combinations of the last two coordinate values, the heuristic scans the whole weight matrix with the step n^2 between the accessed weights, i.e., the distance between the successively accessed weights in the memory is n^2

elements, which is very inefficient. A similar situation occurs when the first and the last dimensions are fixed.

To avoid this disadvantage, we propose the following algorithm. Let M^d be an $n \times n$ matrix for every $1 \le d \le s$. Initialize $M^d_{i,j} = 0$ for every $1 \le d \le s$ and $1 \le i, j \le n$. For each vector $e \in X$ and for each $1 \le d \le s$ set $M^d_{e_d,e_{d+1}} = M^d_{e_d,e_{d+1}} + w(e)$ (here we assume that $e_{s+1} = e_1$). Now the matrices M^d can be used for the first iteration of every ROM run.

When applying this technique, only one full matrix scan is needed for the heuristic and this scan is successive. There are several other inefficient iterations like fixing of the last three coordinates but their influence on the algorithm's performance is negligible.

2.6 Data Structures

In some cases data structure plays the key role in an algorithm's theoretical efficiency (see, e.g., [55] and references there). In other cases it does not change the theoretical time complexity of an algorithm but it is still worth a separate discussion. Below we consider several data structures for GTSP algorithms.

2.6.1 GTSP Tour Storage

It is a non-trivial question how one should store a GTSP solution. The most common approach is to store a sequence of vertices in the visiting order. It was used in [95, 100] and many others. The advantages of this method are simplicity, compactness (it requires only an integer array of size m) and quickness of the weight calculation. The disadvantages are difficulty in some tour modifications (observe that moving one vertex requires up to m operations) and absence of a trivial way to check the tour correctness. Sliding along the tour is easy in this representation but requires some additional checks.

Another tour representation, random-key, was used in [96]. It represents the tour as a sequence of real numbers $x_1, x_2, \ldots, x_m$; the ith number x_i corresponds to the ith cluster C_i of the problem. The integer part $\lfloor x_i \rfloor$ of

the number is the vertex index within the cluster C_i and the fractional part $x_i - \lfloor x_i \rfloor$ determines the position of the cluster in the tour—the clusters are ordered according to these fractional parts, in ascending order. The main advantage of random-key tours is that almost any sequence of numbers represent a correct tour; one only needs to ensure that $1 \leq \lfloor x_i \rfloor \leq |C_i|$ for every i. It is also relatively easy to implement some modifications of the tour. The disadvantages are difficulty in sliding along the tour and the high cost of the tour weight calculation.

We propose a new tour representation which is base on double-linked lists. In particular, we store three integer arrays of size m: *prev*, *next* and *vertices*, where $prev_i$ is the cluster preceding the cluster C_i in the tour, $next_i$ is the cluster succeeding the cluster C_i in the tour, and $vertices_i$ is the vertex within cluster C_i. There are several important advantages of this representation. Unlike other approaches, it naturally represents the cycle which simplifies the algorithms. Consider, e.g., a typical local search implementation (Algorithm 1): the algorithm smoothly slides along the tour until no

Algorithm 1 A typical implementation of a local search based on the double-linked list tour representation. In this example the algorithm preforms as few iterations as possible to ensure that the tour is a local minimum.

```
Set the current cluster X ← 1.
Set the counter t ← m.
while t > 0 do
  if there exist some improvements for the current cluster X then
    Update the tour accordingly.
    Update the counter t ← m.
  else
    Decrease the counter t ← t − 1.
  Move to the next cluster X ← next_X.
```

improvement is found for exactly one loop. Observe that one does not need the concept of position when using this tour representation; it is possible to use cluster index instead. In this context the procedure of tour rotation becomes meaningless; one can simply consider any cluster as the first cluster in the tour. Moreover, it allows one to find a certain cluster in $O(1)$ time; we use it, e.g., to start the CO calculations from the smallest cluster with no extra effort. Our representation clearly splits the cluster order and the

vertex selection; note that some algorithms do not require the information on the vertex selection while some others do not modify the cluster order. It is useful that linked lists allow quick removing and inserting elements. To turn the tour backwards, one only has to swap *prev* and *next*. Observe that this tour representation is deterministic, i.e., each GTSP tour has exactly one representation in this form. If the problem is symmetric, every tour $(prev, next, vertices)$ has exactly one clone $(next, prev, vertices)$.

The main disadvantage of this representation is that it takes three times more space than the sequence of vertices. In fact, implementation of many algorithms do not require backward links. In this case one can avoid using the *prev* array and hence use only two m-elements arrays. When necessary, one can quickly restore the *prev* array according to *next*.

Note that a similar tour representation was used in [101].

2.6.2 GTSP Weights Storage

Another important decision is how to store the weights of a GTSP instance. There are two obvious solutions of this problem:

1. Store a two dimensional matrix M of size $n \times n$ as follows: $M_{i,j} = w(V_i \to V_j)$. Note that this data structure stores $\sum_{i=1}^{m} |C_i|^2$ redundant weights.

2. Store $m(m-1)$ matrices, one matrix $M^{X,Y}$ of size $|X| \times |Y|$ per every pair of distinct clusters X and Y.

If we have a pair of vertices and we need to get the weight between them, it is obviously better to use the first approach. However, if we need to use many weights between two clusters (consider, e.g., calculation of the smallest weight between clusters X and Y: $w_{\min}(X \to Y)$), the second approach is preferable. Indeed, in the first approach we have to use something like M_{X_i,Y_j}, i.e., look for the absolute index of every vertex in X and Y. In the second approach we just find the matrix $M^{X,Y}$ and then use it like this: $M^{X,Y}_{i,j}$. Observe that the second approach provides a sequential access to the weight matrix which is very friendly with respect to the computer architecture, see Section 2.5.2.

Our experimental analysis shows that the second approach improves the performance of some algorithms approximately twice. However, it is not efficient for some other algorithms which behave as a TSP heuristic, i.e., consider only one vertex in every cluster. We decided to use both approaches in our implementations, i.e., to store the weights in a single matrix and, in addition, to store a matrix for every pair of clusters.

2.7 Conclusion

Several aspects of optimization heuristic design and analysis are discussed in this chapter. A lot of attention is paid to the questions of test bed selection. Observe that a typical heuristic does not provide any solution quality guarantee and, hence, experimental evaluation is vastly important.

We consider two examples of test bed generation. For MAP, we systematized the existing instance families. For one of these instance families we have successfully applied probabilistic analysis in order to estimate the exact solution values. Note that for most of instances of this type our estimation is really precise.

There exist several speed-up approaches applicable to virtually any optimization heuristic. One of these approaches in preprocessing. Observe that almost any algorithm hugely depends on the input size. Hence, even a small decrease of the instance size may significantly reduce the running time of a heuristic. We show an example of GTSP preprocessing which removes some vertices and/or edges from an instance if they may not be included in the optimal solution. Our experiments confirm the success of this technique.

At last, we discuss some aspects related to implementation details of an algorithm. In turns out that a simple transformation of an algorithm may significantly speed it up. We provide an example of a very successful optimization of MAP construction heuristics. In addition, we discuss the efficiency of several data structures. We show that selecting a proper data structure may often improve and simplify an algorithm.

Chapter 3

Local Search Algorithms for GTSP

While GTSP is a very important combinatorial optimization problem and is well-studied in many aspects, researches still did not pay enough attention to GTSP specific local search and mostly use simple TSP heuristics with basic adaptations for GTSP. This section aims at thorough and deep investigation of the neighborhoods specific for GTSP and algorithms that can explore these neighborhoods quickly.

We formalize the procedure of adaptation of a TSP neighborhood for GTSP and propose efficient algorithms to explore the obtained neighborhoods. We also generalize all other existing and some new GTSP neighborhoods. Apart from these theoretical results, we also provide the results of a thorough experimental analysis to compare the proposed algorithms implementations and find out which neighborhoods are the most efficient in practice.

Note that some neighborhoods were used in [96, 95, 101], but they were not systematized or analyzed in detail.

We introduce a classification of GTSP neighborhoods. We divide all the neighborhoods into three classes:

1. Cluster Optimization neighborhoods are the neighborhoods which preserve the cluster order in the tour. This class is discussed in Section 3.1.

2. TSP neighborhoods are the neighborhoods produced from the TSP

ones. They usually perform some global rearrangements in the cluster order. In Section 3.2.2 we show that there exist several ways to adapt a TSP neighborhood for GTSP and propose a number of improvements to make these adaptations fast. We thoroughly investigate possible adaptations of the state-of-the-art TSP Lin-Kernighan heuristic in Section 3.3.

3. Fragment Optimization neighborhoods include only tours which are different from the original one in at most some small tour fragment. Neighborhoods of this type were not widely used before. In Section 3.4, we propose two efficient algorithms for these neighborhoods.

In order to compare the efficiency of different neighborhoods and implementations, a series of experiments is conducted in Section 3.5.

3.1 Cluster Optimization

In this section we discuss GTSP neighborhoods which preserve the order of clusters in the tour. In other words, these neighborhoods may only vary the vertices within certain clusters. The virtually smallest neighborhood of this type is

$$N_{\mathrm{L}}(T,i) = \{T_1 \to T_2 \to \ldots \to T_{i-1} \to T'_i \to T_{i+1} \to T_{i+2} \to T_m \to T_1 : \; T'_i \in \mathit{Cluster}(T_i)\}\,.$$

Its size is $|N_{\mathrm{L}}(T,i)| = |\mathit{Cluster}(T_i)|$ and it takes $O(s)$ operations to explore it. One can extend it for two or more clusters: $N_{\mathrm{L}}(T,I)$, where I is a set of cluster indices. The size of such neighborhood $|N_{\mathrm{L}}(T,I)| = \prod_{i\in I} |\mathit{Cluster}(T_i)|$. Observe that while the set I contains no neighbor indices, i.e., if $i \in I$ then $i-1, i+1 \notin I$, it takes only $O(|I|s)$ operations to explore it. If $I = \{i, i+1\}$, the neighborhood $N_{\mathrm{L}}(T,I)$ changes its structure. Now it takes $O(s^2)$ operations to explore it. One may assume that, if $I = \{i, i+1, \ldots, i+k-1\}$, the time complexity of the local search is $O(s^k)$. However, we will show that it remains quadratic for any fixed $k < m$.

Consider the case when $k = m$, i.e., when the vertices are optimized in all the clusters of the tour. This is the most powerful neighborhood of this type

and we call it *Cluster Optimization*. In fact, there is an exact algorithm CO that finds the optimal vertex selection for the whole solution in $O(n\gamma s)$ time. In other words, given a fixed cluster order, it finds the best cycle through these clusters.

CO was introduced by Fischetti, Salazar-González and Toth [25] (see its detailed description also in [23]) and used in [51, 87, 92] and others. It is based on the shortest path algorithm for acyclic digraphs (see, e.g., [9]).

Let $T = T_1 \to T_2 \to \ldots \to T_m \to T_1$ be the given tour and $\mathcal{T}_i = \mathit{Cluster}(T_i)$ for every i. The algorithm builds a layered network $G_{\mathrm{CO}} = (V_{\mathrm{CO}}, E_{\mathrm{CO}})$, where $V_{\mathrm{CO}} = V \cup \mathcal{T}_1'$ is the set of the GTSP instance vertices extended by a copy $\mathcal{T}_1'$ of the cluster $\mathcal{T}_1$, and E_{CO} is a set of edges in the digraph G_{CO}. An edge $x \to y \in E_{\mathrm{CO}}$ exists if there exists i such that $x \in \mathcal{T}_i$ and $y \in \mathcal{T}_{i+1}$ (assume $\mathcal{T}_{m+1} = \mathcal{T}_1'$). The weight of the edge $x \to y$ is $w(x \to y)$. For each vertex $v_1 \in \mathcal{T}_1$ and its copy $v_1' \in \mathcal{T}_1'$, the algorithm finds the shortest (v_1, v_1')-path in G_{CO}. It selects the shortest (v_1, v_1')-path which represents the best vertex selection within the given cluster sequence. A formal procedure based on the dynamic programming approach is presented in Algorithm 2. Note that there is no need to repeat the search several times since it finds

Algorithm 2 Cluster Optimization. Basic implementation.

Require: Tour $T = T_1 \to T_2 \to \ldots \to T_m \to T_1$, where $|\mathit{Cluster}(T_1)| = \gamma$.
  Let $\mathcal{T}_i = \mathit{Cluster}(T_i)$ for every i.
  for all $r \in \mathcal{T}_1$ and $v \in \mathcal{T}_2$ **do**
    Set $p_{r,v} \leftarrow (r \to v)$.
  for $i \leftarrow 3, 4, \ldots, m$ **do**
    for all $r \in \mathcal{T}_1$ and $v \in \mathcal{T}_i$ **do**
      Set $p_{r,v} \leftarrow p_{r,u} + (u \to v)$, where $u \in \mathcal{T}_{i-1}$ is selected to minimize $w\big(p_{r,u} + (u \to v)\big)$.
  return $p_{r,v} + (v \to r)$, where $r \in \mathcal{T}_1$ and $v \in \mathcal{T}_m$ are selected to minimize $w\big(p_{r,v} + (v \to r)\big)$.

the local minimum after the first run.

3.1.1 Cluster Optimization Refinements

Several improvements can noticeably reduce the running time of CO.